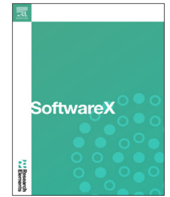
SoftwareX 20 (2022) 101238

Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx



Original software publication

TAFFO: The compiler-based precision tuner

Daniele Cattaneo [a,*], Michele Chiari [a], Giovanni Agosta [a], Stefano Cherubin [b]

[a] DEIB – Politecnico di Milano, Italy
[b] Edinburgh Napier University, United Kingdom

ARTICLE INFO

Article history:
Received 29 April 2021
Received in revised form 28 May 2022
Accepted 15 October 2022

Keywords:
Mixed precision
Compiler
Approximate computing

ABSTRACT

We present TAFFO, a framework that automatically performs precision tuning to exploit the performance/accuracy trade-off. In order to avoid expensive dynamic analyses, TAFFO leverages programmer annotations which encapsulate domain knowledge about the conditions under which the software being optimized will run. As a result, TAFFO is easy to use and provides state-of-the-art optimization efficacy in a variety of hardware configurations and application domains. We provide guidelines for the effective exploitation of TAFFO by showing a typical example of usage on a simple application, achieving a speedup up to 60% at the price of an absolute error of 3.53×10^{-5}. TAFFO is modular and based on the solid LLVM technology, which allows extensibility to improved analysis techniques, and comprehensive support for the most common precision-reduced data types and programming languages. As a result, the TAFFO technology has been selected as the precision tuning tool of the European Training Network on Approximate Computing.

© 2022 The Author(s). Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

Code metadata

Current code version	0.3
Permanent link to code/repository used for this code version	https://github.com/ElsevierSoftwareX/SOFTX-D-21-00084
Legal Code License	MIT License
Code versioning system used	git
Software code languages, tools, and services used	C++
Compilation requirements, operating environments & dependencies	CMake, LLVM
If available Link to developer documentation/manual	https://github.com/TAFFO-org/TAFFO/tree/master/doc
Support email for questions	daniele.cattaneo@polimi.it

1. Motivation and significance

Approximate Computing is an increasingly popular approach to achieve large performance and energy improvements in error-tolerant applications [1,2]. This class of techniques aims at trading off computation accuracy for performance and energy. Within Approximate Computing, a key issue is to perform each computation on the most energy and performance-efficient data type that allows to preserve the desired minimum accuracy. This non-trivial task is usually performed manually by embedded systems programmers, and in general by software developers that need to achieve high performance with limited resources. However, this operation is error-prone and tedious, especially when large code bases are involved. Thus, a significant research effort has been spent over the recent years to build compiler-based tools to support or entirely replace the programmer effort [3].

In this work, we present TAFFO, an autotuning framework that aims at optimising the selection of data types in C and C++ programs, particularly by replacing floating point operations with equivalent ones based on other representations, including fixed point. TAFFO has been proven to enable major speedups on most error-tolerant classes of applications when targeting embedded microcontrollers that lack hardware support for floating point

* Corresponding author.
E-mail addresses: Daniele.Cattaneo@polimi.it (Daniele Cattaneo), Michele.Chiari@polimi.it (Michele Chiari), Giovanni.Agosta@polimi.it (Giovanni Agosta), stefanix@acm.org (Stefano Cherubin).
URLs: https://heaplab.deib.polimi.it (Daniele Cattaneo), https://heaplab.deib.polimi.it (Michele Chiari), https://heaplab.deib.polimi.it (Giovanni Agosta).

https://doi.org/10.1016/j.softx.2022.101238
2352-7110/© 2022 The Author(s). Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

operations, but can also help improve performance and energy on more high-end platforms. TAFFO is implemented as a set of plug-ins for the industry-grade LLVM compiler framework, enabling its deployment in most modern systems.

2. Related works

According to the classification given by Cherubin and Agosta in [3], in the state of the art other precision tuning tools similar to TAFFO are CRAFT [4], Rosa [5] and Daisy [6]. Other tools focus only on one aspect of precision tuning – such as verification of error constraints, or the generation of a data type assignment – and do not provide a comprehensive applicative solution.

CRAFT is a precision tuning framework based on modifying a compiled binary. Through a binary search, an initial set of data type assignments is successively refined to produce a final recommendation. The precision/performance tradeoff is evaluated dynamically by instrumenting the modified binary at each step of the search. CRAFT is exclusively based on dynamic analyses, and is limited to applications targeting the Intel x86 processor architecture. The data types supported by CRAFT are single and double precision floating point.

Rosa is a source-to-source compiler that tunes a program depending on a contract-based specification language that includes the precision requirement of each function. It is limited to programs written in Scala, and uses the Z3 solver [7]. Daisy is a refined version of Rosa that – among other improvements – employs the dReal solver [8] instead of Z3. Rosa and Daisy are limited by the restriction to the Scala language and their usage of a DSL for describing precision requirements. Additionally, they support only single and double precision floating point data types.

Amongst tools that are based on LLVM, the most relevant one is Precimonious [9]. Precimonious computes the most efficient precision mix within an error threshold for a C program. It supports single precision, double precision and 80-bit floating point data types, and the search for the best precision mix is performed with the delta-debugging algorithm [10]. While Precimonious is similar to TAFFO in its use of LLVM, it employs dynamic analyses. Additionally, Precimonious is based on a dated version of LLVM, which limits its usefulness.

We can notice that most precision tuning tools target floating point data types, overlooking other representations. This is appropriate in the high-performance-computing context, but limits applicability in edge computing applications. TAFFO fills this gap, providing an increased set of supported data types – including fixed point. Additionally, static analyses are less common than dynamic analyses. In fact, TAFFO is unique amongst LLVM-based precision tuning tools in exploiting only static analyses.

3. Software description

TAFFO tackles all the challenges of precision tuning [3], and it does so by using safe and deterministic static analyses.

The user is expected to annotate the source code to provide information on the dynamic value range for the input variables and on the scope of the optimization. In general, to achieve the best results, the optimization scope should be a mathematically intensive computational kernel. The annotations to insert depends on the input data to the program, therefore the typical user of TAFFO is a domain expert who has access to the required information. A static data flow analysis propagates the value ranges to all the intermediate values in the program and determines the set of variables that need to be changed in type. This analysis is able to operate across function calls and loops if required.

Based on the fine-grained description of the dynamic value ranges produced by the data flow analyses, TAFFO performs the *data type allocation*. For each dynamic value range, a constraint is derived on the data type such instruction can use. At this point, TAFFO determines the data type to assign to each variable automatically. Two different approaches are available for this task. The first one exploits a local best-fit algorithm that performs adjacent similar type coalescing. A customisable cost function can take into account the overhead introduced by type cast operations only, which varies depending on the target architecture. A second more complex algorithm [11] builds a partial mathematical model of the program that describes the variation in execution time and output error for a given architecture depending on the data type selection. This model is fed into an integer-linear-programming constraint solver to select the optimal data types for each variable that must be optimized. This new approach requires a more through architectural model and is more effective for embedded platforms, while the simpler local best-fit algorithm is more effective on superscalar architectures.

Once the data type has been decided, TAFFO performs the code conversion on the LLVM-IR. Whenever an equivalent instruction (or pattern of instructions) is not immediately available — e.g. in the case of a function call to a generic function — TAFFO implements a two-way best-effort approach. In case of called functions whose definition lives within the same file, TAFFO creates an ad-hoc version of the callee. In the rest of unknown cases, TAFFO rejects the proposed data type and locally uses the original one. Appropriate type-cast operations are inserted if required.

Additionally, in beta versions of TAFFO, in the case of well-known functions, equivalent fixed point code can be generated on-demand [12].

Finally, TAFFO performs a functional and performance estimation of the conversion's benefits via static analysis techniques. It is worth to mention that both the cost function from the data type allocation, and the performance estimator from this last stage require an architectural model of the target machine. In absence of such model, TAFFO uses default values which may be suboptimal for the target architecture.

3.1. Software architecture

TAFFO operates during code compilation as a plug-in for the LLVM compiler infrastructure. This design decision creates several benefits.

Portability The technique is agnostic with respect to source language and target architecture.

Fine-Grained Data type is allocated to each individual operation rather than declared variables.

Compatibility May be combined with other optimisation techniques.

TAFFO accepts as input LLVM intermediate code representation, and it produces the same format as output. The LLVM-IR-equivalent version of a C/C++ program can be obtained via CLANG. A plain un-modified version of the CLANG front-end is also able to parse the additional annotations provided by the user in the source code. These annotations will later be processed by the initial stages of TAFFO. An example of how these annotations appear in the source code is shown in Listing 1.

Annotations can be placed on any variable declaration, and contain information about the value range of the variable itself, and additional directives that affect TAFFO's operation. Detailed documentation on the syntax and the semantics of annotations is found in the TAFFO project repository.[1]

[1] https://github.com/TAFFO-org/TAFFO/blob/master/doc/CommandLineReference.md

```
1  float a __attribute((annotate("scalar(range(-10,10))"))); float b __attribute((annotate("scalar()")));
```

Listing 1: Example of annotated C code where the programmer is requesting the transformation of variables a and b to a fixed point type.

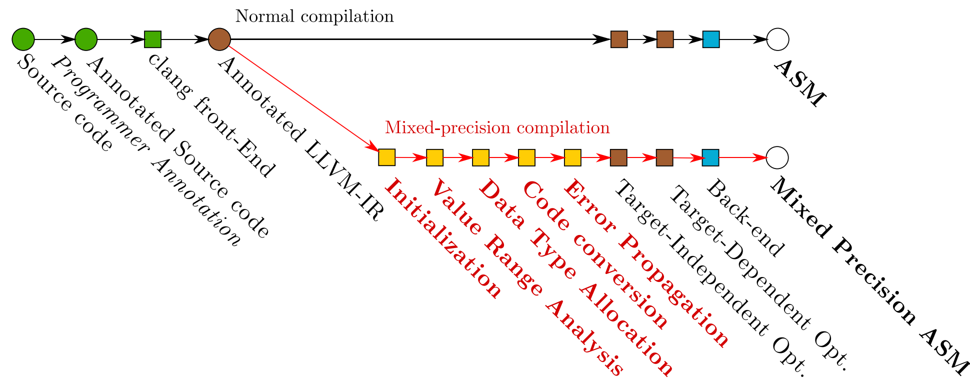


Fig. 1. Outline of the compilation pipeline using the CLANG compiler front-end with and without TAFFO. We highlight with red arrows the TAFFO pipeline stages. Green elements refer to source code and the compiler front-end. Brown elements refer to passes of the optimizer; yellow elements correspond to the passes inserted by TAFFO. Finally, the blue element represents the compiler back-end. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

TAFFO is shipped as a set of LLVM passes, i.e. elementary units of the compiler pipeline. In particular, the execution flow runs through five stages: *Initialization*, *Requirement Analysis*, *Data Type Allocation*, *Code Conversion*, *Delta Estimation*. The outline of the compilation pipeline with and without TAFFO is shown in Fig. 1. The passes can be inserted in any point of the optimization pipeline, but usually they are executed immediately after the frontend. Other normalization and analysis passes are automatically scheduled by LLVM as required.

Initialization. The Initialization pass processes the user annotations, and determines the scope of the transformation which will be performed by later passes — in other words which instructions are affected by the type change of the annotated variables. This process is performed through a reverse depth-first iteration in the data flow graph.

Requirement analysis. This pass performs a fine-grained requirements analysis meant to augment the existing LLVM-IR with additional information. The most important task performed is a Value Range Analysis (VRA) based on Interval Analysis [13], and a simplified symbolic execution framework.

The data type allocation pass. The Data Type Allocation pass decides which data type should be used for each intermediate value. It is designed to avoid overflow. Secondarily, it attempts to minimise accuracy loss and performance degradation due to type cast overhead.

The data types supported by TAFFO are single and double precision floating point, *bfloat16* and fixed point types of arbitrary size and point position. The set of types allowed in the optimized program can be adjusted at compile time.

The code conversion pass. The code conversion and supplementary code generation pass modifies the program code in order to enforce the usage of the data types determined by the previous passes of TAFFO. It preserves code semantics within the given value ranges. Whenever the conversion of an unknown external procedure is required, the data type is retained to the original version.

The error propagation pass. This pass performs an additional code analysis which estimates the error in the output with respect to the original code. The error analysis is based on affine number theory [14,15] in order to handle cancellation errors.

4. Illustrative example

In order to illustrate the functionalities of TAFFO, we show how to optimize an example application, whose source code is reported in Listing 2. This application computes an optical blur (*bokeh*) on a pre-existing image in `ppm` format. The core of the application consists of a convolution of the source image with a fixed-size kernel image, producing in output the blurred picture. Similar image convolution algorithms are widespread not only in the image editing field, but they also find application in the machine-learning field as a pipeline for image recognition.

At a fundamental level, TAFFO provides an eponymous command line program, `taffo`, for compiling a C or C++ program by performing precision tuning optimizations. A standard invocation of `clang` or `gcc` can be replaced with an invocation to `taffo` to enable such optimizations. However, using TAFFO is equivalent to using `clang` when compiling a program without any annotations.

In general, TAFFO requires annotations to be placed on every floating point variable declaration that shall be subject to precision tuning. Therefore, the first step in modifying a program to enable TAFFO's optimizations is to identify such variables and annotate them using an empty annotation (`__attribute__((annotate("scalar()")))`). These annotations do not need to contain any range information.

In the example program, such annotations were placed on the `kernel` matrix, which stores the convolution kernel, and on the two dynamically allocated arrays containing the input and output images respectively. Since the kernel computation process represents only a small fraction of the total execution time of the tool, its floating point variables were not annotated. TAFFO will take care of inserting the proper casts automatically to handle the mixup of annotated and non-annotated variables. Typically, it is not needed to optimize an entire program with TAFFO to obtain speedups, just the main computational kernel suffices.

In addition to deciding which variables will be considered for optimization, the programmer must specify the dynamic value range of input variables. By *input variables* we intend those variables whose value at runtime depends purely on external factors. In the example program, we position the range annotations on the output variables of the `fscanf` input call.

Finally, the range and error analyses in TAFFO require the user to specify one or more *output* variables using a `target`

```c
#define RADIUS (8)
#define K_SIDE (RADIUS*2+1)
float kernel[K_SIDE][K_SIDE]
 __attribute__((annotate("scalar()")));
#define GET_PIX(image, w, h, x, y) \
 ((image)+(y)*(w)*3+(x)*3)
#define MIN(a, b) ((a) < (b) ? (a) : (b))
#define MAX(a, b) ((a) > (b) ? (a) : (b))
#define CLAMP(min, v, max) \
 (MAX((min), MIN((v), (max))))

void bokeh(float *res, float *image,
      int w, int h)
{
 for (int y=0; y<h; y++)
  for (int x=0; x<w; x++)
   for (int ky=0; ky<K_SIDE; ky++)
    for (int kx=0; kx<K_SIDE; kx++) {
     int oy = y - (K_SIDE-1)/2 + ky;
     int ox = x - (K_SIDE-1)/2 + kx;
     if (oy < 0 || oy >= h
       || ox < 0 || ox >= w)
      continue;
     GET_PIX(res,w,h,ox,oy)[0]+=
      GET_PIX(image,w,h,x,y)[0]
      *kernel[ky][kx];
     GET_PIX(res,w,h,ox,oy)[1]+=
      GET_PIX(image,w,h,x,y)[1]
      *kernel[ky][kx];
     GET_PIX(res,w,h,ox,oy)[2]+=
      GET_PIX(image,w,h,x,y)[2]
      *kernel[ky][kx];
    }
}

void compute_kernel(void)
{
 for (int y=0; y<K_SIDE; y++)
  for (int x=0; x<K_SIDE; x++) {
   kernel[y][x] = 0.0;
  }

 float step = 1.0 / 32.0;
 float mass = M_PI * RADIUS * RADIUS;
 for (float y=0; y<K_SIDE; y+=step)
  for (float x=0; x<K_SIDE; x+=step) {
   float cx =
    x - (float)(K_SIDE)/2.0 + 0.5*step;
   float cy =
    y - (float)(K_SIDE)/2.0 + 0.5*step;
   if (cx*cx + cy*cy < RADIUS*RADIUS)
    kernel[(int)y][(int)x] +=
     (step*step) / mass;
  }
}

int read_ppm(FILE *fp, float **image,
       int *w, int *h)
{
 char buf[10];
 uint64_t max __attribute__((annotate
  ("scalar(range(255,255)disabled)")));

 fscanf(fp, "%s", buf);
 if (strcmp(buf, "P3") != 0)
  return 0;
 fscanf(fp, "%d%d%"PRIu64, w, h, &max);
 *image = calloc(sizeof(float), (*w)*(*h)*3);
 for (int y=0; y<*h; y++) {
  for (int x=0; x<*w; x++) {
   for (int p=0; p<3; p++) {
    uint64_t tmp __attribute__((annotate
     ("scalar(range(0,255)disabled)")));
    fscanf(fp, "%"PRIu64, &tmp);
    GET_PIX(*image, *w, *h, x, y)[p] =
     (float)tmp / (float)max;
   }
  }
 }
 return 1;
}

int write_ppm(FILE *fp, float *image,
        int w, int h)
{
 fprintf(fp, "P3\n%d%d\n%d\n",
     w, h, 0x7FFFFFFF);
 for (int y=0; y<h; y++) {
  for (int x=0; x<w; x++) {
   for (int p=0; p<3; p++) {
    uint64_t cc = (uint64_t)(CLAMP(0.0,
     GET_PIX(image, w, h, x, y)[p], 1.0)
     * 0x7FFFFFFF);
    fprintf(fp, "%"PRIu64"", cc);
   }
  }
  fprintf(fp, "\n");
 }
 return 1;
}

int main(int argc, char *argv[])
{
 if (argc != 3)
  return 1;

 float *image
  __attribute__((annotate("scalar()")));
 int w, h;
 FILE *fp = fopen(argv[1], "r");
 if (!read_ppm(fp, &image, &w, &h))
  return 1;
 fclose(fp);

 float *res __attribute__
  ((annotate("target('res')scalar()")));
 res = calloc(sizeof(float), w*h*3);
 compute_kernel();
 bokeh(res, image, w, h);

 fp = fopen(argv[2], "w");
 write_ppm(fp, res, w, h);
 fclose(fp);

 free(image);
 free(res);
 return 0;
}
```

Listing 2: The listing of the example program illustrated in Section 4, including the annotations required by TAFFO.

annotation. We add this additional annotation on the dynamic array which will contain the output image.

After these simple modifications, the program is ready to be compiled using TAFFO. The command line we use is the following:

```
taffo -O3 bokeh.c -o bokeh_taffo
```

Once the compilation completes, the file `bokeh_taffo` will contain the executable program optimized by TAFFO.

In Fig. 2 we show an example picture as processed both by the unmodified example program, and the same program optimized by TAFFO. The two images appear identical. The maximum difference between the pixel component values of the two images is 3.53×10^{-5}, which is consistent with the two images being identical except for the quantization error introduced by the conversion to fixed point. The execution time is also different

Fig. 2. Output of the example program shown in Listing 2, both in its original form (left), as optimized by TAFFO (right), and the difference between the two images (center). The maximum difference between the pixel component values of the two images is 3.53×10^{-5}, a difference that disappears completely when re-scaled to 8-bit component values.

between the TAFFO optimized version and the unmodified floating point version. This is especially noteworthy on architectures with slow floating-point units. For instance, one run of the unmodified floating-point `kernel` function, as compiled by LLVM Clang 10.0.1 on a ARM1176JZ-F processor clocked at 700 MHz takes 13.4 s, while the optimized version compiled with TAFFO executes in 8.4 s. This amounts to a speedup of 60%.

Additional arguments can be passed to TAFFO to further decrease the precision of the output. As an example, the argument `-Xdta -totalbits -Xdta 8` will instruct TAFFO to prefer 8 bit fixed point numbers. Exhaustive documentation of these options is found in the TAFFO documentation.

5. Impact

Approximate Computing is an emerging field that is quickly gaining traction and attention from industry and research groups, as attested by a number of very recent literature reviews [1–3,16]. Apart from TAFFO, current state-of-the-art tools for precision tuning include Precimonious [9], Daisy [6] and CRAFT [4]. These tools are either based on dynamic analyses (such as Precimonious and CRAFT) or require the use of domain-specific languages (CRAFT). As a result, TAFFO is unique in its ability to be both language-independent (due to its use of LLVM-IR) and fully based on static analyses.

TAFFO has been developed as part of the ANTAREX project [17,18], and is currently the only solution for precision tuning available for the industry-standard LLVM compiler [3]. Furthermore, TAFFO is the precision tuning tool adopted and supported by the European Training Network on approximate computing, APROPOS.[2] This network will train the next generation of approximate computing experts in Europe, and includes 14 institutions from 9 countries. TAFFO is supported by the Italian National Interuniversity Consortium for Informatics (CINI) Workgroup on High Performance Computing as one of its key technologies [19]. As such, TAFFO will be used as the precision tuning tool by the EuroHPC TEXTAROSSA consortium [20], which aims at providing mixed precision support in extreme-scale computing systems. The consortium comprises, beyond CINI, three European HPC centers, as well as the key industrial providers of HPC in Europe. As a result, TAFFO is ideally positioned to become the main tool for mixed precision tuning of applications across a wide range of domains and platforms. In fact, since TAFFO is based on LLVM-IR, which is platform and language independent, it can be used in GPU applications or as a part of an HLS toolchain. The only basic requirement is to introduce the appropriate annotations in the intermediate LLVM-IR code, be it a shader kernel for a GPU or a program meant for HLS synthesis.

The potential impact of TAFFO is reinforced by use-case explorations performed in the field of operating system schedulers [21], activity classification [22] and motor field-oriented control [23]. Thanks to the precision tuning optimization performed by TAFFO, the operating system scheduler state machine update function achieved a speedup up to 80%, the activity classification workload gained a speed-up of approximately 500%, and the algorithm for field-oriented control obtained a speedup of approximately 250%. The effectiveness of TAFFO has also been proven on well-known benchmark suites such as AxBench [24] and PolyBench [25] in works such as [11,26]. In particular, in [12] the usage of TAFFO for optimizing the implementation of trigonometric functions in the benchmarks of the AxBench suite resulted in energy savings of up to 60%.

[2] https://projects.tuni.fi/apropos/

6. Conclusions

In this paper, we described the structure of TAFFO, a precision tuning assistant based on compiler analyses and transformations. We provide an overview of the key software components, and how they tackle the precision tuning challenges. We believe TAFFO represents a valuable addition to any hardware/software co-design toolchain. Its main features — *portability*, *fine granularity*, and *compatibility* — allow it to fit in contexts ranging from HPC to embedded systems.

Although TAFFO's main focus remains fixed point representations, its approach generalises well to new, trending numeric representations such as *bfloat16*. Future releases of TAFFO will explicitly support *bfloat16* and an increased set of numeric representations, parallel applications, GPU-based kernels and High-Level-Synthesis workflows.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work is supported by the European Commission and the Italian Ministry of Economic Development (MISE) under the EuroHPC TEXTAROSSA project (G.A. 956831).

References

[1] Sampson A, Dietl W, Fortuna E, Gnanapragasam D, Ceze L, Grossman D. EnerJ: Approximate data types for safe and general low-power computation. In: Proceedings of the 32nd ACM SIGPLAN conference on programming language design and implementation. New York, NY, USA: ACM; 2011, p. 164–74. http://dx.doi.org/10.1145/1993498.1993518.

[2] Mittal S. A survey of techniques for approximate computing. ACM Comput Surv 2016;48(4):62:1–33. http://dx.doi.org/10.1145/2893356.

[3] Cherubin S, Agosta G. Tools for reduced precision computation: a survey. ACM Comput Surv 2020;53(2). http://dx.doi.org/10.1145/3381039.

[4] Lam MO. CRAFT: Configurable Runtime Analysis for Floating-point Tuning. 2018, https://github.com/crafthpc/craft. [Accessed 07 August 2018].

[5] Darulova E. Rosa, the real compiler. 2015, https://github.com/malyzajko/rosa. [Accessed 07 August 2018].

[6] Darulova E, Horn E, Sharma S. Sound mixed-precision optimization with rewriting. In: Proceedings of the 9th ACM/IEEE international conference on cyber-physical systems. 2018, p. 208–19. http://dx.doi.org/10.1109/ICCPS.2018.00028.

[7] De Moura L, Bjørner N. Z3: An efficient SMT solver. In: Proceedings of the theory and practice of software, 14th international conference on tools and algorithms for the construction and analysis of systems. TACAS'08/ETAPS'08, 2008, p. 337–40.

[8] Gao S, Kong S, Clarke EM. dReal: An SMT solver for nonlinear theories over the reals. In: Automated deduction – CADE-24. 2013, p. 208–14.

[9] Rubio-González C, Nguyen C, Nguyen HD, Demmel J, Kahan W, Sen K, et al. Precimonious: Tuning assistant for floating-point precision. In: Proceedings of the international conference on high performance computing, networking, storage and analysis. 2013, p. 27:1–27:12. http://dx.doi.org/10.1145/2503210.2503296.

[10] Zeller A, Hildebrandt R. Simplifying and isolating failure-inducing input. IEEE Trans Softw Eng 2002;28(2):183–200. http://dx.doi.org/10.1109/32.988498.

[11] Cattaneo D, Chiari M, Fossati N, Cherubin S, Agosta G. Architecture-aware precision tuning with multiple number representation systems. In: 2021 58th ACM/IEEE Design Automation Conference. 2021, p. 673–8. http://dx.doi.org/10.1109/DAC18074.2021.9586303.

[12] Cattaneo D, Chiari M, Magnani G, Fossati N, Cherubin S, Agosta G. FixM: Code generation of fixed point mathematical functions. Sustain Comput Inform Syst 2021;29. http://dx.doi.org/10.1016/j.suscom.2020.100478, URL http://www.sciencedirect.com/science/article/pii/S2210537920302018.

[13] Moore RE, et al. Introduction to interval analysis. Siam; 2009.

[14] de Figueiredo LH, Stolfi J. Affine arithmetic: Concepts and applications. Numer Algorithms 2004;37(1):147–58. http://dx.doi.org/10.1023/B:NUMA.0000049462.70970.b6.

[15] Darulova E, Kuncak V. Towards a compiler for reals. ACM Trans Program Lang Syst 2017;39(2):8:1–28. http://dx.doi.org/10.1145/3014426.

[16] Moreau T, San Miguel J, Wyse M, Bornholt J, Alaghi A, Ceze L, et al. A taxonomy of general purpose approximate computing techniques. IEEE Embed Syst Lett 2018;10(1):2–5. http://dx.doi.org/10.1109/LES.2017.2758679.

[17] Cherubin S, Cattaneo D, Chiari M, Agosta G. Dynamic precision autotuning with TAFFO. ACM Trans Archit Code Optim 2020;17(2). http://dx.doi.org/10.1145/3388785.

[18] Silvano C, Agosta G, Barbosa J, Bartolini A, Beccari AR, Benini L, et al. The ANTAREX tool flow for monitoring and autotuning energy efficient HPC systems. In: 2017 International conference on embedded computer systems: architectures, modeling, and simulation. 2017, p. 308–16. http://dx.doi.org/10.1109/SAMOS.2017.8344645.

[19] Aldinucci M, et al. The Italian research on HPC key technologies across EuroHPC. In: Proceedings of the 18th ACM international conference on computing frontiers. New York, NY, USA: Association for Computing Machinery; 2021, p. 178–84. http://dx.doi.org/10.1145/3457388.3458508.

[20] Agosta G, et al. TEXTAROSSA: Towards EXtreme scale Technologies and Accelerators for euROhpc hw/Sw Supercomputing Applications for exascale. In: 2021 24th Euromicro conference on digital system design. 2021, p. 286–94. http://dx.doi.org/10.1109/DSD53832.2021.00051.

[21] Cattaneo D, Di Bello A, Cherubin S, Terraneo F, Agosta G. Embedded operating system optimization through floating to fixed point compiler transformation. In: 21st Euromicro conference on digital system design. 2018, p. 172–6. http://dx.doi.org/10.1109/DSD.2018.00042.

[22] Fossati N, et al. Automated precision tuning in activity classification systems: A case study. In: Proceedings of the 11th workshop on parallel programming and run-time management techniques for many-core architectures / 9th workshop on design tools and architectures for multicore embedded computing platforms. PARMA-DITAM 2020, 2020, http://dx.doi.org/10.1145/3381427.3381432.

[23] Magnani G, Cattaneo D, Chiari M, Agosta G. The Impact of Precision Tuning on Embedded Systems Performance: A Case Study on Field-Oriented Control. In: Bispo Ja, Cherubin S, Flich J, editors. 12th Workshop on parallel programming and run-time management techniques for many-core architectures and 10th workshop on design tools and architectures for multicore embedded computing platforms. Open access series in informatics (OASIcs), vol. 88, Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum Für Informatik; 2021, p. 3:1–3:13. http://dx.doi.org/10.4230/OASIcs.PARMA-DITAM.2021.3, URL https://drops.dagstuhl.de/opus/volltexte/2021/13639.

[24] Yazdanbakhsh A, et al. AxBench: A multiplatform benchmark suite for approximate computing. IEEE Des Test 2017;34(2):60–8. http://dx.doi.org/10.1109/MDAT.2016.2630270.

[25] Yuki T. Understanding PolyBench/C 3.2 kernels. In: International workshop on polyhedral compilation techniques. 2014.

[26] Cherubin S, Cattaneo D, Chiari M, Giovanni A. Dynamic precision autotuning with TAFFO. ACM Trans Archit Code Optim 2020;17(2). http://dx.doi.org/10.1145/3388785.